PHarr XCPC Templates

PHarr

2023 年 7 月 19 日

目录

第一章　编程技巧和基础算法

1.1　读入优化

```
// 快读
int read() {
    int x = 0, f = 1, ch = getchar();
    while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
    if (ch == '-') f = -1, ch = getchar();
    while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
        ch = getchar();
    return x * f;
}
// 关闭同步
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

1.2　C++ 标准输入输出

```
// 设置输出宽度为 x
cout << setw(x) << val;

// 设置保留小数位数 x , 并四舍五入
cout << fixd<< setprecision(x) << val;

//按进制输出
cout << bitset<10>(i); // 二进制
cout << oct << i; // 八进制
cout << dec << i; // 十进制
cout << hex << i; // 十六进制

```

```
// 设置填充符
cout << setw(10) << 1234; // 默认是空格
cout << setw(10) << setfill('0') << 1234; // 设置填充符
//左侧填充
cout<<setw(10)<<setfill('0')<<setiosflags(ios::left)<<123;
//右侧填充
cout<<setw(10)<<setfill('0')<<setiosflags(ios::right)<<123;

// 单个字符
ch = cin.get();
cout.put(ch);

// 指定长度字符串读入
cout.get(str,80,'a');// 字符串 字符个数 终止字符

// 整行读入
getlin( cin , s );
```

1.3 倍增

天才 *ACM*

给定一个整数 M，对于任意一个整数集合 S，定义“校验值”如下:

从集合 S 中取出 M 对数 (即 2×M 个数，不能重复使用集合中的数，如果 S 中的整数不够 M 对，则取到不能取为止)，使得“每对数的差的平方”之和最大，这个最大值就称为集合 S 的“校验值”。

现在给定一个长度为 N 的数列 A 以及一个整数 T。我们要把 A 分成若干段，使得每一段的“校验值”都不超过 T。求最少需要分成几段。

1. 初始化`p = 1 , r = l = 1`
2. 求出`[l,r+p]`这一段的校验值，若校验值小于等于 T 则`r+=p,p*=2`，否则`p/=2`
3. 重复上一步知道 p 的值变为 0 此时的 r 即为所求

```
#include<bits/stdc++.h>
#define int long long
using namespace std;

int read() {
    int x = 0, f = 1, ch = getchar();
```

```
7     while ((ch < '0' || ch > '9') && ch != '-') ch = getchar();
8     if (ch == '-') f = -1, ch = getchar();
9     while (ch >= '0' && ch <= '9') x = (x << 3) + (x << 1) + ch - '0',
          ch = getchar();
10    return x * f;
11 }
12
13 const int N = 5e5+5;
14 int a[N] , b[N];
15 int n , m , t , res;
16 int query( int l , int r ){
17    if( r > n ) return 1e19;
18    for( int i = l ; i <= r ; i ++ ) b[i] = a[i];
19    sort( b + l , b + r + 1);
20    int ans = 0;
21    for( int i = l , j = r , t = 1 ; t <= m && i < j ; t ++ , i ++ , j
          -- )
22        ans += ( b[i] - b[j] ) * ( b[i] - b[j] );
23    return ans;
24 }
25
26 void solve(){
27    n = read() , m = read() , t = read() , res = 0;
28    for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
29    for( int l = 1 , r = 1 , p; l <= n ; l = r + 1 ){
30        p = 1 , r = l;
31        while( p ){
32            if( query( l , r + p ) <= t ) r += p , p *= 2;
33            else p /= 2;
34        }
35        res ++;
36    }
37    cout << res << "\n";
38 }
39
40 int32_t main() {
41    for( int T = read(); T ; T -- ) solve();
42    return 0;
```

```
43 }
```

1.4 语法杂项

1.4.1 define 与 typedef

typedef是用来给类型定义别名，是编译器处理的

#define是字面上进行宏定义用的，是在预处理阶段使用的

```
1 #define STRING char * //宏定义
2 STRING name , sign;//声明
3 char * name , sign;//会被替换成这种的结果,只有 name 是指针
4 // 所以定义类型时候应该回避 #define 而采用typedef
5 typedef char * STRING;
6 char * name , * sign;//会被替换成这种结果
```

1.4.2 fill 数组填充

```
1 // 一维数组
2 int a[5];
3 fill( a , a + 5 , 3 );
4
5 // 二维数组
6 int a[5][4];
7 fill( a[0] , a[0] + 5 * 4 , 6 );
8
9 // vector
10 vector<int> a;
11 fill( a.begin() , a.end() , 9 );
```

1.5 Lambda 表达式

以下内容绝大部分使用与c++14及更新的标准

Lambda 的组成部分是

```
1 [capture] (parameters) mutable -> return-type {statement};
```

首先caputre是捕获列表可以从所在代码块中捕获变量。

什么都不写[]就是不进行任何捕获，[=]是值捕获，[&]是引用捕获，值捕获不能修改变量的值，引用捕获可以。特别的，如果值捕获希望在函数内部修改可以使用mutable关键字

同时捕获列表也可以单独针对某一个变量[a]、[&a]分别是值捕获和引用捕获。当然也可以混用[=,&a]对所有变量值捕获，但a除外，a是引用捕获。

然后就是parameters参数列表和statement函数主体，这里与普通的函数没有区别。

-> return-type，函数范围值类型，如果不写可以自动推断，但是如果有多个return且返回类型不同就会CE

c++14之后可以用auto来自动的把函数赋值给变量，c++11中则需要自己写

1.6 MT 19937

mt19937是一个很便捷的随机数生成算法，在 c++11中使用非常便捷

```
mt19937 rd(seed) ; // 这样就填入了一个随机数种子
rd(); // 这样就会返回一个随机数
mt19937_64 rd(); // 相同用法，不过返回是一个 64 位整形
```

1.7 常用常数

在<math.h>库中有一些常用的参数

```
#if defined _USE_MATH_DEFINES && !defined _MATH_DEFINES_DEFINED
    #define _MATH_DEFINES_DEFINED
    #define M_E        2.71828182845904523536   // e
    #define M_LOG2E    1.44269504088896340736   // log2(e)
    #define M_LOG10E   0.434294481903251827651  // log10(e)
    #define M_LN2      0.693147180559945309417  // ln(2)
    #define M_LN10     2.30258509299404568402   // ln(10)
    #define M_PI       3.14159265358979323846   // pi
    #define M_PI_2     1.57079632679489661923   // pi/2
    #define M_PI_4     0.785398163397448309616  // pi/4
    #define M_1_PI     0.318309886183790671538  // 1/pi
    #define M_2_PI     0.636619772367581343076  // 2/pi
    #define M_2_SQRTPI 1.12837916709551257390   // 2/sqrt(pi)
    #define M_SQRT2    1.41421356237309504880   // sqrt(2)
    #define M_SQRT1_2  0.707106781186547524401  // 1/sqrt(2)
#endif
```

但是<math.h>并没有默认定义_USE_MATH_DEFINES，所以用之前需要先定义（在万能头下貌似已经被定义过了），在开头加上#define _USE_MATH_DEFINES即可

1.8 有用的库函数

1. `fabs(x)` 用于计算浮点数的绝对值
2. `exp(x)` 计算 e^x
3. `log(x)` 计算 $\ln(x)$
4. `__lg(x)` 计算 $\lfloor \log_2(x) \rfloor$
5. `__gcd(x,y)` 计算 $gcd(x,y)$, 不过在 C++17 引入了`gcd(x,y)`,`lcm(x,y)`
6. `ceil(x)` 返回 $\lceil x \rceil$
7. `floor(x)` $\lfloor x \rfloor$
8. `next_permutation(begin,end)` 将 $[begin, end)$ 变为下一个排列，如果已经是最后一个排列就返回 0
9. `prev_permutation(begin,end)` 将 $[begin, end)$ 变为上一个排列，如果已经是最后一个排列就返回 0
10. `shuffle(begin,end,gen)` 打乱 $[begin, end)$，`gen`是一个随机数生成器（参考 mt19937）
11. `is_sorted(begin,end)` 判断是否升序排序
12. `max(l)`,`min(l)` 对于数组或列表返回最大最小值，例`max({x,y,z})`
13. `exp2(x)` 计算 2^x
14. `log2(x)` 计算 $\log_2(x)$
15. `hypot(x,y)` 计算 $\sqrt{x^2+y^2}$
16. `rotate(iterator begin, iterator middle, iterator end)`作用是把序列中`begin`和`end`连起来，然后再从`middle`处断开，`middle`作为新的`begin`

1.9 交互题

这里说的交互题只是 STDIO 交互题

交互题往往是不会限制运行时间的，一般是通过限制与 oj 的交换次数。对于 IO 交换的题目，要注意的是在每一次输出后都必须要**刷新输出缓冲**后才可以读入。下面介绍几种语言如何刷新缓冲。

1. C `fflush(stdout)`
2. C++ `fflush(stdout)` 或者`cout << flush` 或者用`cout << endl`输出换行也会自动刷新

3. Java `System.out.flush()`

4. Python `stdout.flush()`

1.10 Windows 下的对拍器

```
:again
data.exe > data.in
std.exe < data.in > std.out
test.exe < data.in > test.out
fc std.out test.out
if not errorlevel 1 goto again
```

1.11 Linux 下的对拍器

```
#!/bin/bash
while true; do
    ./data > data.in
    ./std <data.in >std.out
    ./code <data.in >code.out
    if diff std.out code.out; then
        printf "AC\n"
    else
        printf "Wa\n"
        exit 0
    fi
done
```

第二章　数据结构

2.1　并查集

```
1  // 初始化
2  for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
3  // 查找
4  int getFa( int x ){
5      if( fa[x] == x ) return x;
6      return fa[x] = getFa( fa[x] );
7  }
8  // 合并
9  void merge( int x , int y ){
10     fa[getFa(x) ] = getFa(y);
11 }
12
13 /*
14  * 下面是一种按秩合并的写法
15  * 简单来说fa[x]<0表示该点为根结点, 当x为根节点时 fa[x] = -size[x]
16  */
17 class dsu{
18 private:
19     vector<int> fa;
20 public:
21     dsu( int n = 1 ){
22         fa = vector<int>( n+1 , -1 ) , fa[0] = 0;
23     }
24     int getfa( int x ){
25         if( fa[x] < 0 ) return x;
26         return fa[x] = getfa( fa[x] );
27     }
28     void merge( int x , int y ){
```

```
29          x = getfa(x) , y = getfa(y);
30          if( x == y ) return ;
31          if( fa[x] > fa[y] ) swap( x , y );
32          fa[x] += fa[y] , fa[y] = x;
33      }
34      bool check( int x , int y ){
35          x = getfa(x) , y = getfa(y);
36          return ( x == y );
37      }
38  };
```

2.2 链式前向星

链式前向星又名邻接表，其实现在我已经几乎不会再手写链式前向星而是采用vector来代替

```
1  vector<int> e[N];// 无边权
2  vector< pair<int,int> > e[N]; 有边权
3
4  e[u].push_back(v);// 加边(u,v)
5  e[u].push_back( { v, w } ); //加有权边 (u,v,w)
6  // 无向边 反过来再做一次就好
7
8  for( auto v : e[u] ){ // 遍历
9  }
10 for( auto [ v , w ] : e[u] ) { // 遍历有权边
11 }
```

2.3 Hash

2.3.1 Hash 表

对数字的 hash

```
1  for( int i = 1 ; i <= n ; i ++ ) b[i] = a[i]; // 复制数组
2  sort( b + 1 , b + 1 + n ) , m = unique( b + 1 , b + 1 + n ) - b;// 排
       序去重
3  for( int i = 1 ; i <= n ; i ++ )//hash
4      a[i] = lower_bound( b + 1 , b + 1 + m , a[i] ) - b;
```

除此之外，如果更加复杂的 hash 全部使用unordered_map容器

2.4 栈

2.4.1 包含 Min 函数的栈

一个支持 $O(1)$ 的 $push(), pop(), top(), getmin()$ 的栈

在维护栈的同时维护一个栈来保存历史上每个时刻都最小值

```
struct MinStack{
    stack<int> a , b;
    void push( int x ){
        a.push(x);
        if( b.size() ) b.push( min( x , b.top() ) );
        else b.push(x);
    }
    void pop(){
        a.pop() , b.pop();
    }
    int top(){
        return a.top();
    }
    int getMin(){
        return b.top();
    }
};
```

2.4.2 单调栈

用来 $O(n)$ 的维护出每一个点左侧第一个比他高的点

```
int h[N] , l[N]; // h[i] 用来记录每一个点的高度 l[i] 记录每一个左侧第一个比 i 高的点的位置
stack<int> stk;

h[0] = INF; //为了便于处理把 0 处理为正无穷
stk.push( h[0] );
for( int i = 1 ; i <= n ; i ++ ){
    while( h[i] >= h[ stk.top() ] ) stk.pop();
    l[i] = stk.top() , stk.push(i);
}
```

2.5 ST 表

ST 表解决的问题是没有修改且查询次数较多（10^6）的区间最值查询

f[i][j]表示 i 向后 2^j 个数的最大值

所以 f[i][j] = max(f[i][j-1] , f[i + (1 << j-1)][j-1]

询问首先计算出数最大的 x 满足 $2^x \le r-l+1$

这样的话 $[l,r]=[l,l+2^x-1]\cup[r-2^x+1,r]$

```
// LOJ10119
const int N = 1e6+5 , logN = 20;
int a[N] , log_2[N] , f[N][ logN + 5 ];
int32_t main() {
    int n = read() , m = read();
    for( int i = 1 ; i <= n ; i ++ )
        a[i] = read();
    log_2[0] = -1; // 这样初始化可以使得 log_2[1] = 0

    for( int i = 1 ; i <= n ; i ++ ) // O(n) 预处理边界条件 和 log2(i)
        f[i][0] = a[i] , log_2[i] = log_2[i>>1] + 1;

    for( int j = 1 ; j <= logN ; j ++ )
        for( int i = 1; i + ( 1  << j ) - 1 <= n ; i ++ )
            f[i][j] = max( f[i][j-1] , f[ i + ( 1 << j - 1 ) ][j-1] );

    for( int l , r , s ; m ; m -- ){
        l = read() , r = read() , s = log_2[ r - l + 1 ];
        printf("%d\n" , max( f[l][s] , f[ r - ( 1 << s ) + 1 ][s] ));
    }
    return 0;
}
```

2.6 树状数组

2.6.1 单点修改，区间查询

```
struct BinaryIndexedTree{
#define lowbit(x) ( x & -x )
    int n;
    vector<int> b;
```

```
5
6       BinaryIndexedTree( int n ) : n(n) , b(n+1 , 0){};
7       BinaryIndexedTree( vector<int> &c ){ // 注意数组下标必须从 1 开始
8           n = c.size() , b = c;
9           for( int i = 1 , fa = i + lowbit(i) ; i <= n ; i ++ , fa = i +
                lowbit(i) )
10              if( fa <= n ) b[fa] += b[i];
11      }
12      void add( int i , int y ){
13          for( ; i <= n ; i += lowbit(i) ) b[i] += y;
14          return;
15      }
16
17      int calc( int i ){
18          int sum = 0;
19          for( ; i ; i -= lowbit(i) ) sum += b[i];
20          return sum;
21      }
22  };
```

2.6.2 区间修改，单点查询

这里用线段树维护一下差分数组就好

```
1  // op == 1 [l,r] 加上 val
2  // op == 2 查询位置 l 的值
3  int32_t main() {
4      n = read() , m = read();
5      vector<int> t(n+1);
6      for( int i = 1 , x = 0 , lst = 0; i <= n ; i ++ ) x = read() , t[i
          ] = x - lst , lst = x ;
7      BinaryIndexedTree B(t);
8      for( int op , l , r , val; m ; m -- ){
9          op = read();
10         if( op == 1 ) l = read() , r = read() , val = read() , B.add(
              l , val ) , B.add( r + 1 , - val );
11         else l = read() , printf("%d\n" , B.calc(l) );
12     }
13     return 0;
```

```
}
```

2.7 分块

```
// https://loj.ac/p/6280
#include <bits/stdc++.h>
using namespace std;
#define int long long

int read() {...}
class decompose {
private:
    struct block {
        int l, r, sum, tag;// tag是单点修改时的懒惰标记
        vector<int> val;

        block(int l, int r) : l(l), r(r) {
            sum = tag = 0;
            val = vector<int>();
        }
    };
    int len;
    vector<block> part;
    vector<int> pos;
public:
    decompose(vector<int> &v) {
        len = v.size();
        int t = sqrt(len);
        pos = vector<int>(len + 1);
        for (int i = 1; i <= t; i++) // 预处理区间信息
            part.emplace_back((i - 1) * t + 1, i * t);
        if (part.back().r < len) // 处理结尾零散部分
            part.emplace_back(part.back().r + 1, len);
        for (int i = 1, j = 0; i <= len; i++) {
            if (i > part[j].r) j++;
            part[j].val.emplace_back(v[i - 1]);
            part[j].sum += v[i - 1], pos[i] = j;
        }
```

```
35      }
36
37      int getSum(int l, int r) {
38          int sum = 0;
39          for (int i = pos[l]; i <= pos[r]; i++) {
40              if (part[i].l >= l && part[i].r <= r) sum += part[i].sum;
41              else
42                  for( auto j = max(l, part[i].l) - part[i].l; j <= min(
                        r, part[i].r) - part[i].l ; j++ )
43                      sum += part[i].val[j] + part[i].tag;
44          }
45          return sum;
46      }
47
48      void update(int l, int r, int d) {
49          for (int i = pos[l]; i <= pos[r]; i++) {
50              if (part[i].l >= l && part[i].r <= r){
51                  part[i].tag += d;
52                  part[i].sum += part[i].val.size() * d;
53              }
54              else
55                  for (int j = max(l, part[i].l) - part[i].l; j <= min(r
                        , part[i].r) - part[i].l; j++)
56                      part[i].val[j] += d, part[i].sum += d;
57          }
58      }
59  };
60
61
62  int32_t main() {
63      int n = read();
64      vector<int> a(n);
65      for (auto &i: a) i = read();
66      decompose p(a);
67      for (int opt, l, r, c, sum; n; n--) {
68          opt = read(), l = read(), r = read(), c = read();
69          if (opt == 0)
70              p.update(l, r, c);
```

```
71          else {
72              c++, sum = p.getSum(l, r), sum = (sum % c + c) % c;
73              printf("%lld\n", sum);
74          }
75      }
76      return 0;
77  }
```

2.8 ODT

```
1   class ODT{
2   private:
3       struct Node{ // 节点存储结构
4           int l , r;
5           mutable int val;
6           Node( int l , int r = 0 , int val = 0 ) :
7                   l(l) , r(r) , val(val){};
8           bool operator < ( Node b ) const {
9               return l < b.l;
10          }
11          int len() const{
12              return r - l + 1;
13          }
14      };
15      int len;
16      set<Node> s;
17  public:
18      // 两种构造函数
19      ODT( const int & n , const vector<int> & a ){
20          len = n;
21          int t = 1;
22          for( int v : a )
23              s.insert( Node( t , t , v ) ) , t ++;
24      }
25      ODT( const int & n , const int a[] ){
26          len = n;
27          for( int i = 1 ; i <= n ; i ++ )
28              s.insert( Node( i , i , a[i] ) );
```

```
29      }
30
31      auto split( int x ){ // 区间分裂
32          if( x > len ) return s.end();
33          auto it = --s.upper_bound( Node( x ) );
34          if( it->l == x ) return it;
35          int l = it->l , r = it->r , v = it->val;
36          s.erase(it);
37          s.insert( Node( l , x-1 , v ) );
38          return s.insert( Node( x , r , v ) ).first;
39      }
40      void assign( int l , int r , int v ){ // 区间赋值 平推操作
41          auto itr = split( r + 1 ) , itl = split( l );
42          s.erase( itl , itr );
43          s.insert( Node( l , r , v ) );
44      }
45      void add( int l , int r , int x ){ // 区间修改
46          auto itr = split( r+1 ) , itl = split( l );
47          for( auto it = itl ; it != itr ; it ++ )
48              it->val +=x;
49      }
50      int rank( int l , int r , int x ){ // 求区间第 x 大
51          auto itr = split(r+1) , itl = split(l);
52          vector< pair<int,int> > v; // first 表示 num , second 表示 cnt
53          for( auto it = itl ; it != itr ; it ++ )
54              v.push_back({ it->val , it->len() } );
55          sort( v.begin() , v.end() );
56          for( auto [ num , cnt ] : v ){
57              if( cnt < x ) x -= cnt;
58              else return num;
59          }
60      }
61      void merge( int l , int r ){ // 区间合并 推平
62          auto itr = split(r+1) , itl = split(l);
63          vector<Node> cur;
64          for( auto it = itl ; it != itr ; it ++ ){
65              if( cur.empty() || it->val != cur.back().val )
66                  cur.push_back( *it );
```

```
67              else cur.back().r = it->r;
68          }
69          s.erase( itl , itr );
70          for( auto it : cur )
71              s.insert( it );
72          return;
73      }
74      int calP( int l , int r , int x , int y ){ // 求区间 x 次方之和
75          auto itr = split(r+1) , itl= split(l);
76          int ans = 0;
77          for( auto it = itl ; it != itr ; it ++ )
78              ans = ( ans + power(it->val,x,y) * it->len()%y ) % y;
79          return ans % y;
80      }
81  };
```

2.9 线段树

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define int long long
5
6  const int N = 5e5+5;
7  int n , m , a[N];
8
9  struct Node{
10     int l , r , value , add;
11     Node * left , * right;
12     Node( int l , int r , int value , int add , Node * left , Node *
           right ) :
13             l(l) , r(r) , value(value) , add(add) , left(left) , right
                   (right) {};
14 } * root;
15
16 int read() {...}
17 // 建树
18 Node * build( int l , int r ){
```

```
19      if( l == r ) return new Node( l , r , a[l] , 0 , nullptr , nullptr
            );
20      int mid = ( l + r ) >> 1;
21      Node * left = build( l , mid ) , * right = build( mid + 1 , r );
22      return new Node( l , r , left->value+right->value,0,left , right);
23  }
24  // 标记
25  void mark( int v , Node * cur ){
26      cur -> add += v;
27      cur -> value += v * ( cur->r - cur->l + 1 );
28  }
29  // 标记下传
30  void pushdown( Node * cur ){
31      if( cur -> add == 0 ) return;
32      mark( cur -> add , cur -> left ) , mark( cur -> add , cur -> right
            );
33      cur -> add = 0;
34  }
35  // 修改
36  void modify( int l , int r , int v , Node * cur ){
37      if( l > cur -> r || r < cur -> l ) return;
38      if( l <= cur -> l && r >= cur -> r ){
39          mark( v , cur );
40          return;
41      }
42      pushdown( cur );
43      int mid = ( cur -> l + cur -> r ) >> 1;
44      if( l <= mid ) modify( l , r , v , cur -> left );
45      if( r > mid ) modify( l , r , v , cur -> right );
46      cur -> value = cur ->left->value + cur ->right->value;
47      return ;
48  }
49  // 查询
50  int query( int l , int r , Node * cur ){
51      if( l <= cur->l && r >= cur ->r ) return cur->value;
52      pushdown( cur );
53      int mid = ( cur->l + cur->r ) >> 1 , res = 0;
54      if( l <= mid ) res += query( l , r , cur->left );
```

```
    if( r > mid ) res += query( l , r , cur->right );
    return res;

}

int32_t main(){
    n = read() , m = read();
    for( int i = 1 ; i <= n ; i ++ ) a[i] = read();
    root = build( 1 , n );
    for( int i = 1 , opt , l , r ; i <= m ; i ++ ){
        opt = read() , l = read() , r = read();
        if( opt == 1 ) modify( l , r , read() , root );
        else printf("%lld\n" , query( l , r , root ) );
    }
    return 0;
}
```

2.10 差分

离散化差分

```
// 离散化差分这里我用map实现
map<int,int> b;

void update( int l , int r , int v ){
    b[l] += v , b[r+1] -= v;
}

// 求和
for( auto it = b.begin() ; next(it) != b.end() ; it = next(it) )
    next(it)->second += it->second;
```

2.10.1 二维前缀和、差分

```
// 二维前缀和
b[i][j] = b[i-1][j]+b[i][j-1]-b[i-1][j-1] + a[i][j];

// 求 (x1,y1) ~ (x2,y2) 和
b[x2][y2] - b[x2][y1-1] - b[x1-1][y2] + b[x1-1][y1-1];
```

第三章　图论

3.1　拓扑排序

在一个 DAG 中，将图中的顶点以线性的方式排序，使得对于任何一条 u 到 v 的有向边，u 都可以出现在 v 的前面

拓扑序判环如果图中已经没有入度为零的点但是依旧有点时，即说明图中存在环。

拓扑序判链如果求拓扑序的过程中队列中同时存在两个及以上的元素，说明拓扑序不唯一，不是一条链

字典序最大、最小的拓扑序把 Kahn 中队列换成是大根堆、小根堆实现的优先队列就好

3.1.1　DFS 算法

```
1  vector<int> e[N]; // 邻接表
2  vector<int> topo; // 存储拓扑序
3  set<int> notInDeg;// 储存没有入读的点
4  int vis[N];
5
6  bool dfs( int u ){
7      vis[u] = 1;
8      for( auto v : e[u] ){
9          if( vis[v] )  return 0;
10         if( !dfs(v) ) return 0;
11     }
12     topo.push_back(u);
13     return 1;
14 }
15
16 bool topSort(){
17     if( notInDeg.empty() ) return 0 ;
18     for( int u : notInDeg ){
19         if( !dfs(u) ) return 0;
20     }
```

```
21      reverse( topo.begin() , topo.end() );
22      return 1;
23  }
```

3.1.2 Kahn 算法

```
1  vector<int> e[N]; // 邻接表
2  vector<int> topo; // 存储拓扑序
3  int inDeg[N]; // 记录点的当前入度
4
5  bool topSort(){
6      queue<int> q;
7      for( int i = 1 ; i <= n ; i ++ )
8          if( inDeg[i] == 0 ) q.push(i);
9      while( q.size() ){
10         int u = q.front() ; q.pop();
11         topo.push_back(u);
12         for( auto v : e[u] ){
13             if( --inDeg[v] == 0 ) q.push(v);
14         }
15     }
16     return topo.size() == n;
17 }
```

3.2 最近公共祖先 (LCA)

3.2.1 向上标记法

向上标价法最差的复杂度是 $O(N)$ 的，适用于求 LCA 次数少的情况，代码非常好写

```
1  \\ luogu P3379
2  const int N = 5e5+5;
3  int n , m , sta , dep[N] , fa[N];
4  vector<int> e[N];
5
6  int read() {...}
7  void dfs( int x ){
8      for( auto v : e[x] ){
9          if( v == fa[x] ) continue;
```

```
10          dep[v] = dep[x] + 1 , fa[v] =x;
11          dfs(v);
12      }
13  }
14
15  int lca( int x , int y ){
16      if( dep[x] < dep[y] ) swap( x , y );
17      while( dep[x] > dep[y] ) x = fa[x];
18      while( x != y ) x = fa[x] , y = fa[y];
19      return x;
20  }
21
22  int32_t main() {
23      n = read() - 1 , m = read() , sta = read();
24      for( int u , v ; n ; n -- )
25          u = read(),v = read() , e[u].push_back(v) , e[v].push_back(u);
26      dep[sta] = 0 , fa[sta] = sta;
27      dfs( sta );
28      for( int x , y ; m ; m -- ){
29          x = read() , y = read();
30          cout << lca(x,y) << endl;
31      }
32      return 0;
33  }
```

3.2.2 倍增 LCA

值得注意的是向上倍增的过程中`for( int i = t ; i >= 0 ; i --)` 不能写错

```
1  const int N = 5e5+5;
2  int n , m , sta , logN , dep[N] , fa[N][20];
3  vector<int> e[N];
4  int read() {...}
5
6  void dfs( int x ){
7      for( auto v : e[x] ){
8          if( dep[v] ) continue;
9          dep[v] = dep[x] + 1 , fa[v][0] = x;
10         for( int i = 1 ; i <= logN ; i ++ )
```

```
11              fa[v][i] = fa[ fa[v][i-1] ][i-1];
12          dfs(v);
13      }
14  }
15  int lca( int x , int y ){
16      if( dep[x] > dep[y] ) swap( x , y );
17      for( int i = logN ; i >= 0 ; i -- )
18          if( dep[ fa[y][i] ] >= dep[x] ) y = fa[y][i];
19      if( x == y ) return x;
20      for( int i = logN ; i >= 0 ; i -- ){
21          if( fa[x][i] != fa[y][i] ) x = fa[x][i] , y = fa[y][i];
22      }
23      return fa[x][0];
24  }
25
26  int32_t main() {
27      n=read()-1 , m=read() , sta=read() , logN =(int)log2(n) + 1;
28      int k = n;
29      for( int u , v ; n ; n -- )
30          u=read() , v=read() , e[u].push_back(v) , e[v].push_back(u);
31      dep[sta] = 1;
32      dfs( sta );
33      for( int x , y ; m ; m -- ){
34          x = read() , y = read();
35          cout << lca(x,y) << endl;
36      }
37      return 0;
38  }
```

3.2.3 Tarjan LCA

3.2.4 RMQ 求 LCA

一次 dfs 求出树的深度和欧拉序。区间[l,r]中的深度的最小值就是 lca。区间最值用 st 表来维护即可。

3.3 最短路

3.3.1 Floyd

```
int dis[N][N];
for( int i = 1 ; i <= n ; i ++ )
    for( int j = 1 ; j < i ; j ++ )
        dis[i][j] = dis[j][i] = inf;

for( int u , v , w ; m ; m -- )
    u = read() , v = read() , w = read() , dis[u][v] = dis[v][u] = w;
for( int k = 1 ; k <= n ; k ++ )
    for( int i = 1 ; i <= n ; i ++ )
        for( int j = 1 ; j < i ; j ++ )
            f[i][j] = f[j][i] = min( f[i][j] , f[i][k] + f[k][j] );
```

3.3.2 Dijkstra

复杂度是 $O(m\log(n))$

```
void dij(){
    for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
    dis[sta] = 0;
    priority_queue< pair<int,int> , vector<pair<int,int>> , greater<
       pair<int,int>> > q;
    q.push( { 0 , sta } );
    while( q.size() ){
        int u = q.top().second ; q.pop();
        if( vis[u] ) continue;
        vis[u] = 1;
        for( auto [v,w] : e[u] )
            if( dis[v] > dis[u] + w ){
                dis[v] = dis[u] + w;
                q.push( {dis[v] , v } );
            }
    }
}
```

3.3.3 Bellman-Ford

复杂度是 $O(km)$

```
bool bellmanFord(){ // 返回是否有最短路
    for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
    dis[sta] = 0;
    bool flag;
    for( int i = 1 ; i <= n ; i ++ ){
        flag = 0;
        for( int u = 1 ; u <= n ; u ++ ){
            if( dis[u] == inf ) continue; // 如果当前点和起点没有联通，就无法进行松弛操作
            for( auto [v,w] : e[u] ){
                if( dis[v] <= dis[u] + w ) continue;
                dis[v] = dis[u] + w , flag = 1;// 记录时候进行松弛操作
            }
        }
        if( !flag ) break;
    }
    return flag;
}
```

3.3.4 SPFA

没有准确复杂度，下限是 $O(m\log(n))$，上限是 $O(nm)$

```
int dis[N] , cnt[N];
vector< pair<int,int> > e[N];
bitset<N> vis;

bool spfa(){
    for( int i = 1 ; i <= n ; i ++ ) dis[i] = inf;
    queue< int > q;
    dis[sta] = 0 , vis[sta] = 1 , q.push(sta);
    for( int u ; q.size() ; ){
        u = q.front() , q.pop() , vis[u] = 0;
        for( auto [ v , w ] : e[u] ){
            if( dis[v] <= dis[u] + w ) continue;
            dis[v] = dis[u] + w ;
            cnt[v] = cnt[u] + 1; // 记录最短路经过了几条边
```

```
15              if( cnt[v] >= n ) // 最短路最长是 n-1
16                  return false; // 此时说明出现了 负环
17              if( !vis[v] ) vis[v] = 1 , q.push(v);
18          }
19      }
20      return true;
21  }
```

3.4 最小生成树

3.4.1 Kruskal

Kruskal 总是维护无向图的最小生成深林。

```
1  // Luogu P3366
2  const int N = 5005;
3  int n , m , fa[N] , cnt = 0 , sum;
4
5  int read() {...}
6
7  int getfa( int x ){...}
8
9  void merge( int x , int y ){...} // 并查集合并
10
11 int32_t main(){
12     n = read() , m = read();
13     for( int i = 1 ; i <= n ; i ++ ) fa[i] = i;
14     vector<tuple<int,int,int>> e(m);
15     for( auto & [ w , u , v ] : e )
16         u = read() , v = read() , w = read();
17     sort( e.begin() , e.end() );
18     for( auto[ w , u , v ] : e ){
19         if( getfa(u) == getfa(v) ) continue;
20         merge( u , v ) , sum += w , cnt ++;
21         if( cnt == n - 1 ) break;
22     }
23     if( cnt == n - 1 ) cout << sum << "\n";
24     else cout << "orz\n";// 图不联通
25     return 0;
```

```
}
```

3.5 树

3.5.1 树的深度

```
dep[sta] = 1;

void dfs( int u ){
    for( auto v : e[u] ){
        if( dep[v] ) continue;
        dep[v] = dep[u] + 1;
        dfs( v );
    }
}

dfs(sta);
```

3.5.2 二叉树还原

```
struct Node {
    int v;
    Node *l, *r;
    Node(int v, Node *l, Node *r) : v(v), l(l), r(r) {};
};
// 根据中序、后序还原二叉树
Node *build(vector<int> mid, vector<int> suf) {
    int v = suf.back();
    Node *l = nullptr, *r = nullptr;
    int t;
    for (t = 0; t < mid.size(); t++)
        if (mid[t] == v) break;
    auto midll = mid.begin();
    auto midlr = mid.begin() + t;
    auto midrl = mid.begin() + t + 1;
    auto midrr = mid.end();
    auto sufll = suf.begin();
    auto suflr = suf.begin() + t;
```

```
19      auto sufrl = suf.begin() + t;
20      auto sufrr = suf.end() - 1;
21
22      auto midl = vector<int>(midll, midlr);
23      auto midr = vector<int>(midrl, midrr);
24      auto sufl = vector<int>(sufll, suflr);
25      auto sufr = vector<int>(sufrl, sufrr);
26
27      if (!midl.empty()) l = build(midl, sufl);
28      if (!midr.empty()) r = build(midr, sufr);
29
30      return new Node(v, l, r);
31  }
```

3.5.3 树的 DFS 序和欧拉序

DFS 序中一个点会出现一次，欧拉序会出现两次，两者都是 dfs 时经过的点的顺序，欧拉序中的第二次出现就是点结束搜索是回溯时的顺序

```
1  // 求DFS序
2  vector<int> dfsSort;
3  vector<int> e[N];
4  bitset<N> vis;
5
6  void dfs( int x ){
7      dfsSort.push_back(x) , vis[x] = 1;
8      for( auto it : e[x] ){
9          if( vis[it] ) continue;
10         dfs( it );
11     }
12 }
13 // 求欧拉序
14 vector<int> eulerSort;
15 vector<int> e[N];
16 bitset<N> vis;
17
18 void dfs( int x ){
19     eulerSort.push_back(x) , vis[x] = 1;
20     for( auto it : e[x] ){
```

```
21          if( vis[it] ) continue;
22          dfs( it );
23      }
24      eulerSort.push_back(x);
25  }
```

3.5.4 树的重心

对于size[i]表示每个点子树大小,max_part(x)表示删去x后最大的子树的大小,max_part取到最小值的点p就是树的重心

```
1  void dfs( int u ){
2      vis[u] = size[u] = 1;
3      for( auto v : e[u] ){
4          if( vis[v] ) continue;
5          dfs(v);
6          size[x] += size[v];
7          max_part = max( max_part , size[v] );
8      }
9      max_part = max( max_part , n - size[u] );
10     if( max_part > ans )
11         ans = max_part , pos = u;
12 }
```

3.5.5 树上前缀和

设sum[i]表示节点i到根节点的权值总和。

如果是点权，x,y路径上的和为sum[x]+sum[y]-sum[lca]-sum[fa[lca]]

```
1  // Loj 2491
2  // 一颗树根节点是 1 , 点权就是深度的 k 次方
3  // m次询问,每次问(u,v)路径上点权之和
4  // k 每次都不同但是取值范围只有[1,50]
5  #define int long long
6  const int N = 3e5+5 , mod = 998244353;
7  int n , sum[N][55] , fa[N][20] , dep[N] , logN;
8  vector<int> e[N];
9
10 int read(){...}
11
```

```
12 void dfs( int x ){
13     for( auto v : e[x] ){
14         if( v == fa[x][0] ) continue;
15         dep[v] = dep[x] + 1 , fa[v][0] = x;
16         for( int i = 1 , val = 1; i <= 50 ; i ++ )
17             val = val * dep[v]% mod , sum[v][i] = (val + sum[x][i]) %
                   mod;
18         for( int i = 1 ; i <= logN ; i ++ )
19             fa[v][i] = fa[ fa[v][i-1] ][i-1];
20         dfs(v);
21     }
22 }
23
24 int lca( int x , int y ){...}
25
26 int32_t main(){
27     n = read() , logN = (int)log2(n)+1;
28     for( int i = 2 , u , v ; i <= n ; i ++ )
29         u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u
               );
30     dep[1] = 0;
31     dfs( 1 );
32     for( int m = read() , u , v , k , t ; m ; m -- ){
33         u = read() , v = read() , k = read() , t = lca( u , v );
34         cout << (sum[u][k]+sum[v][k]-sum[t][k]-sum[fa[t][0]][k]+2*mod)
               %mod << "\n";
35     }
36     return 0;
37 }
```

如果是边权，x,y路径上的和为sum[x]+sum[y]-2*sum[lca]

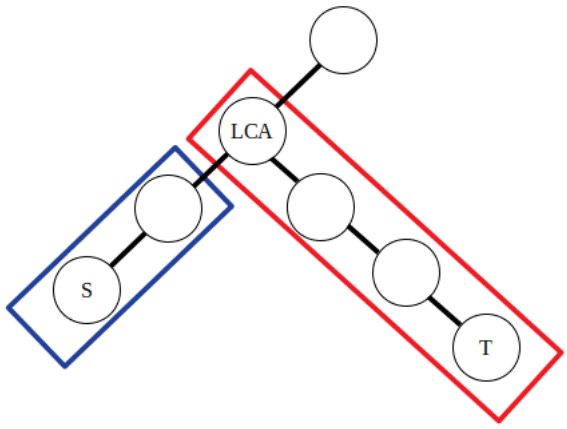
```
1 //LOJ 10134 树上前缀和 边
2 const int N = 1e4+5;
3
4 int n , m , sum[N] , logN , dep[N] , fa[N][15];
5 vector<pair<int,int>> e[N];
6
7 int read(){...}
8
```

```
 9 void dfs( int x ){ // 这里和 lca 极其类似，就是在多维护了一个前缀和
10     for( auto [v,w] : e[x] ){
11         if( dep[v] ) continue;
12         dep[v] = dep[x] + 1 , fa[v][0] = x , sum[v] = sum[x] + w;
13         for( int i = 1 ; i <= logN ; i ++ )
14             fa[v][i] = fa[ fa[v][i-1] ][ i-1 ];
15         dfs(v);
16     }
17 }
18
19 int lca( int x , int y ){...}//这里就是 lca 的板子
20
21 int main(){
22     n = read() , m = read() ,logN = (int)log2(n)+1;
23     for( int i = 2 , u , v , w ; i <= n ; i ++ )
24         u = read() , v = read() , w = read() , e[u].push_back( {v,w} )
               , e[v].push_back( {u,w} );
25     dep[1] = 1 , dfs(1);
26     for( int u , v ; m ; m -- ){
27         u = read() , v = read();
28         cout << sum[u] + sum[v] - 2 * sum[ lca(u,v) ] << "\n";
29     }
30     return 0;
31 }
```

3.5.6 树上差分

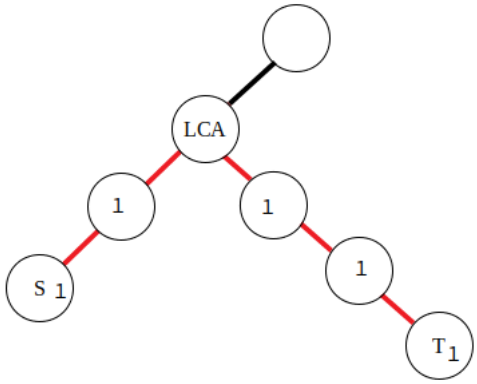
树上差分就是对树上某一段路径进行差分操作，树上差分分为**点差分**和**边差分**

这里的差分数组用d[i]表示，其值是a[fa[i]]-a[i] **点差分**



点差分实际上是要对链分成两条链来操作，如果要对(S,T)路径上点权加p
则要d[s] += p , d[t] += p , d[lca] -= p , d[fa[lca]] -= p

边差分



因为直接对边差分比较困难，所以要把边权移动到边上的子节点上，如果要对(S,T)路径上边权加p，则要d[s] += p , d[t] += p , d[lca] -= 2 * p

```
1  // Luogu P3128 边差分模板
2  #include<bits/stdc++.h>
3  using namespace std;
4
5  const int N = 5e4+5;
6  int n , m , fa[N][30] , logN , dep[N] ,p[N] , res;
7  vector<int> e[N];
8
9  int read(){...}
10
```

```
11 void dfs( int x ) {...} // 和 lca 的dfs完全相同
12
13 int lca( int x , int y ){...} // lca
14
15 void getRes( int x ){
16     for( auto v : e[x] ){
17         if( fa[x][0] == v ) continue;
18         getRes(v) , p[x] += p[v];
19     }
20     res = max( res, p[x] );
21 }
22
23 int32_t main(){
24     n = read() , m = read() , logN = log2(n)+1;
25     for( int u , v , i = 1 ; i < n ; i ++ )
26         u = read() , v = read() , e[u].push_back(v) , e[v].push_back(u
               );
27     dep[1] = 1;
28     dfs( 1 );
29     for( int i = 1 , u , v , d ; i <= m ; i ++ ){
30         u = read() , v = read() , d = lca( u , v );
31         p[u] ++ , p[v] ++ , p[d] -- , p[ fa[d][0] ] --;
32     }
33     getRes( 1 );
34     cout << res << "\n";
35 }
```

3.6 图论杂项

3.6.1 判断简单无向图图

根据图中的每个点的度可以判断，这个图是不是一个简单无向图（没有重边和自环的无向图）

Havel-Hakimi 定理

1. 对当前数列排序，使其呈递减
2. 从 S[2] 开始对其后 S[1] 个数字-1
3. 一直循环直到当前序列出现负数（即不可简单图化的情况）或者当前序列全为 0（可简单图化）时退出。

```
1  //NC-contest-38105-K
2  int read() {...}
3  priority_queue<int> q; vector<int> ve;
4  int32_t main() {
5      int n = read();
6      for( int i = 1 , x ; i <= n ; i ++ ){
7          x = read() , q.push(x);
8          if( x >= n ) // 如果度数大于 n 一定存在重边或自环
9              cout << "NO\n" , exit(0);
10     }
11     while( 1 ){
12         int k = q.top(); q.pop();
13         if( k == 0 )
14             cout << "YES\n" , exit(0);
15         ve.clear();
16         for( int x ; k ; k -- ){
17             x = q.top() - 1 , q.pop();
18             if( x < 0 )
19                 cout << "NO\n" , exit(0);
20             ve.push_back(x);
21         }
22         for( auto it : ve ) q.push(it);
23     }
24     return 0;
25 }
```

第四章　数学知识

4.1　数论

4.1.1　整除

定义：若整数 b 除以非零整数 a ，商为整数且余数为零我们就说 b 能被 a 整除，或 a 整除 b 记作 $a|b$

性质

1. 传递性，若 $a|b, b|c$，则 $a|c$

2. 组合性，若 $a|b, a|c$ 则对于任意整数 m, n 均满足 $a|mb + nc$

3. 自反性，对于任意的 n，均有 $n|n$

4. 对称性，若 $a|b, b|a$ 则 $a = b$

4.1.2　约数

定义若整数 n 除以整数 x 的余数为 0，即 d 能整除 n，则称 d 是 n 的约数，n 是 d 的倍数，记为 $d|n$ 算数基本定理由算数基本定理得正整数 N 可以写作 $N = p_1^{C_1} \times p_2^{C_2} \times p_3^{C_3} \cdots \times p_m^{C_m}$

分解质因数

分解成 $p_1 \times_2 \times p_3 \times \cdots p_n$ 这种形式

```
vector< int > factorize( int x ){
    vector<int> ans;
    for( int i = 2 ; i * i <= x ; i ++){
        while( x % i == 0 )
            ans.push_back(i) , x /= i;
    }
    if( x > 1 ) ans.push_back(x);
    return ans;
}
```

分解成 $p_1^{k_1} \times p_2^{k_2} \times p_3^{k_3} \times \cdots p_n^{k_n}$

```
vector< pair<int,int> > factorize( int x ){
    vector<pair<int,int>> ans;
    for( int i = 2 , cnt ; i * i <= x ; i ++){
        if( x % i ) continue;
        cnt = 0;
        while( x % i == 0 ) cnt ++ , x /= i;
        ans.push_back( { i , cnt } );
    }
    if( x > 1 ) ans.push_back( { x , 1 } );
    return ans;
}
```

N 的正约数个数为 (Π 是连乘积的符号，类似 $\sum$)

$$(c_1+1)\times(c_2+1)\times\cdots(c_m+1)=\Pi_{i=1}^{m}(ci+1)$$

N 的所有正约数和为

$$(1+p_1+p_1^2+\cdots+p_1^{c_1})\times\cdots\times(1+p_m+p_m^2+\cdots+p_m^{c_m})=\prod_{i=1}^{m}(\sum_{j=0}^{c_i}(p_i)^j)$$

4.1.3 GCD 和 LCM

性质 $a\times b=gcd(a,b)\times(a,b)$

通过性质可以得到最小公倍数的求法就是

```
    int lcm( int x , int y ){
        return a / gcd( x , y ) * b;
    }
```

最大公倍数的求法有**更相减损术**和**辗转相除法**

```
int gcd( int x , int y ){ // 更相减损术
    while( x != y ){
        if( x > y ) x -= y;
        else y -=x;
    }
    return x;
}

int gcd( int x , int y ){ // 辗转相除法
    return b ? gcd( b , a % b ) : a ;
}
```

```
12
13 // 还有一种是直接调用库函数
14 __gcd( a , b );
```

一般情况下直接用库函数，库函数的实现是辗转相除法，如果遇到高精度的话（高精度取模分困难）可以用更相减损术来代替

定理对于斐波那契数列 Feb_i 有 $Feb_{gcd(a,b)} = gcd(Feb_a, Feb_b)$

4.1.4 扩展欧几里得

4.1.5 质数

判断质数

```
1 bool isPrime( int x ){
2     for( int i = 1 ; i * i <= x ; i ++ )
3         if( x % i == 0 ) return 0;
4     return 1;
5 }
```

埃式筛

```
1 vector< int > prime;
2 bitset<N>notPrime;//不是素数
3
4 void getPrimes( int n ){
5     notPrime[1] = notPrime[0] = 1;
6     for( int i = 2 ; i <= n ; i ++ ){
7         if( notPrime[i] ) continue;
8         prime.push_back(i);
9         for( int j = i * 2 ; j <= n ; j += i )
10            notPrime[j] = 1;
11    }
12 }
13
14 // 如果不需要 prime 数组的话可以优化成下面的代码
15 bitset<N>notPrime;//不是素数
16
17 void getPrimes( int n ){
18    notPrime[1] = notPrime[0] = 1;
19    for( int i = 2 ; i * i <= n ; i ++ ){
20        if( notPrime[i] ) continue;
```

```
21          for( int j = i * 2 ; j <= n ; j += i )
22              notPrime[j] = 1;
23      }
24  }
```

欧拉筛

```
1  vector< int > prime;
2  bool notPrime[N];;//不是素数
3
4  void getPrimes( int n ){
5      notPrime[1] = notPrime[0] = 1;
6      for( int i = 2 ; i <= n ; i ++ ){
7          if( !notPrime[i] ) prime.push_back(i);
8          for( auto it : prime ){
9              if( it * i > n ) break;
10             notPrime[ it * i ] = 1;
11             if( i % it == 0 ) break;
12         }
13     }
14 }
```

证明质数有无限个

反证法假设数是 n 个，每个素数是 p_i，令 $P = \Pi_{i=1}^{n} p_i + 1$

因为任何一个数都可以分解成多个质数相乘

所以 P 除以任何一个质数都余 1 ，显然 P 就也是一个质数，与假设矛盾，所以假设错误

所以质数是无限个

性质 2

设 $\pi(n)$ 为不超过 n 的质数个数，则 $\pi(n) \approx \frac{n}{\ln n}$

4.1.6 逆元

费马小定理

$a^{p-1} \equiv 1(mod\ p)$, 其中 p 为素数, 所以 $aa^{p-2} \equiv 1(mod\ p)$

```
1  int inv( int x ) {return pow( x , p - 2 );}
```

$O(n)$ **递推**

```
1  const int N = 1005;
2  int inv[N] = {};
3  void invers(int n,int mod){
4      inv[1] = 1;
```

```
5     for(int i = 2;i <= n;i ++) inv[i] = (p-p/i) * inv[p%i] % p;
6     return ;
7 }
```

4.1.7 扩展欧几里得

裴蜀定理

设 a,b 是不全为零的整数，则存在整数 x,y，使得 $ax+by=\gcd(a,b)$

```
1 int exgcd( int a , int b , int & x , int & y ){
2     if( b == 0 ) { x = 1 , y = 0 ; return a;}
3     int d = exgcd( b , a%b , x , y );
4     int z = x ; x = y ; y = z - y * (a / b);
5     return d;
6 }
```

丢番图方程

$ax+by=c$

定义变量 d,x_0,y_0，调用`d = exgcd(a, b, x0, y0)`。对于方程的特解为

$$(x=\frac{c}{d}x_0,y=\frac{c}{d}y_0)(x=\frac{c}{d}x_0,y=\frac{c}{d}y_0)$$

对于方程的通解为

$$(x=\frac{c}{d}x_0+k\frac{b}{d},y=\frac{c}{d}y_0+k\frac{a}{d}),k\in$$

线性同余方程

$a\times x\equiv b(\mod m)$

线性同余方程等价于 $a\times x-b$ 是 m 的倍数，设为 $-y$ 倍，方程可改写为丢番图方程 $a\times x+m\times y=b$

线性同余方程有解的充要条件 $\gcd(a,m)|b$

在有解时用扩偶求得 x_0,y_0 满足 $a\times x_0+m\times y_0=\gcd(a,m)$，则方程的特解 $x=x_0\times\frac{b}{\gcd(a,m)}$

通解是 $x=x_0\times\frac{b}{\gcd(a,m)}+k\times\frac{m}{\gcd(a,m)},k\in$

```
1 int calc(int a, int b, int m) {
2     int x, y, d;
3     d = exgcd(a, m, x, y);
4     if (b % d) return -1;
5     return x * b / d;
6 }
```

扩展欧几里得求逆元

本质上是解同余方程 $a\times a^{-1}\equiv 1(\mod m)$

```
int inv(int a, int m) {
    int x, y, d;
    d = exgcd(a, m, x, y);
    if (d != 1) return -1;
    return (x % m + m) % m;
}
```

线性同余方程组（中国剩余定理）

设 $m_1, m_2, \ldots, m_n$ 是两两互质的整数。$m = \Pi_{i=1}^{n} m_i, M_i = \frac{m}{m_i}$, t_i 是线性同余方程组 $M_i t_i \equiv 1 (\mod m)$ 的一个解，对于任意的 n 个整数 $a_1, a_2, \ldots, a_n$，方程组

$$\begin{cases} x \equiv a_1 (\mod m_1) \\ x \equiv a_2 (\mod m_2) \\ \quad\vdots \\ x \equiv a_n (\mod m_n) \end{cases}$$

有整数解，解为 $x = \sum_{i=1}^{n} a_i M_i t_i (\mod m)$

```
int CRT(int n, vector<int> a, vector<int> m) {
    int mm = 1, ans = 0;
    for (int i = 1; i <= n; i++) mm = mm * m[i];
    for (int i = 1; i <= n; i++) {
        int M = mm / m[i], t, y;
        exgcd(M, m[i], t, y); // t * M % m[i] = 1;
        ans = (ans + a[i] * M * t % mm) % mm;
    }
    return ans;
}
```

高次同余方程 (BSGS)

$a^x \equiv b (\mod p)$，要求 a, p 互质。求非负整数 x。复杂度 $O(\sqrt{p})$

```
int BSGS(int a, int b, int p) {
    map<int, int> hash;
    b %= p;
    int t = sqrt(p) + 1;
    for (int j = 0, val; j < t; j++) {
        val = b * power(a, j, p) % p;
        hash[val] = j;
    }
    a = power(a, t, p);
    if (a == 0) return b == 0 ? 1 : -1;
```

```
    for (int i = 0, val, j; i <= t; i++) {
        val = power(a, i, p);
        j = hash.find(val) == hash.end() ? -1 : hash[val];
        if (j >= 0 && i * t - j >= 0) return i * t - j;
    }
    return -1;
}
```

4.2 数学杂项

4.2.1 二维向量的叉积

$\vec{A} \times \vec{B} = |A||B|\cos(\alpha) = a_x \times b_y - a_y \times b_x$

虽然叉积的结果是一个标量，但是叉积是有正负的，正负取决于向量夹角的大小。

应用：判断点是否在三角形的内部对于三角形 abc 和一个点 o，$\vec{ao} \times \vec{ab}$ 的正负表示了点 o 在线 ab 的左侧还是右侧。只要按照顺时针方向（或逆时针方向）判断点 o 在三条直线的同一侧，既可以判断点在三角形的内部。

```
class Triangle{
    typedef std::pair<int,int> Vector;
private:
    Vector getVector( int x , int y , int a , int b ){
        return std::pair{ a - x , b - y };
    }
    bool product( Vector a , Vector b ){
        return ( a.first * b.second - a.second * b.first ) > 0;
    }
public:
    int ax , ay , bx ,by , cx , cy;
    Vector ab , bc , ca;
    Triangle( int ax , int ay , int bx , int by , int cx , int cy ):
            ax(ax) , ay(ay) , bx(bx) , by(by) , cx(cx) , cy(cy) ,
            ab( getVector( ax , ay , bx , by ) ),
            bc( getVector( bx , by , cx , cy ) ),
            ca( getVector( cx , cy , ax , ay ) ){};
    Triangle(){};
    bool isInTriangle( int x , int y ){
        Vector ao = getVector( ax , ay , x , y );
        Vector bo = getVector( bx , by , x , y );
```

```
22          Vector co = getVector( cx , cy , x , y );
23          bool f1 = product( ao , ab ) , f2 = product( bo , bc ) , f3 =
               product( co , ca );
24          return ( f1 == f2 && f2 == f3 );
25      }
26  };
```

4.3 组合数学

4.3.1 公式

排列数

$$A_n^n = \frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$$

组合数

$$C_m^n = \frac{m!}{(m-n)! \times n!}$$

组合数性质

1. $C_n^m = C_n^{n-m}$
2. $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
3. $C_n^0 + C_n^1 + \cdots + C_n^n = 2^N$
4. $C_n^0 + C_n^2 + C_n^4 + \cdots = C_n^1 + C_n^3 + C_n^5 \cdots = 2^{n-1}$
5. $C_n^m = \frac{n-m+1}{n} \times C_n^{m-1}$

4.3.2 组合数计算

```
1  // 暴力计算组合数
2  int C(int x, int y) { // x 中 选 y 个
3      y = min(y, x - y);
4      int res = 1;
5      for (int i = x, j = 1; j <= y; i--, j++)
6          res = res * i / j;
7      return res;
8  }
9
10 // 加法递推O(n^2)
11 for( int i = 0 ; i <= n ; i ++ ){
```

```
12      c[i][0] = 1;
13      for( int j = 1 ; j <= i ; j ++)
14          c[i][j] = c[i-1][j] + c[i-1][j-1];
15  }
16  // 乘法递推O(n)
17  c[0] = 1;
18  for( int i = 1 ; i * 2 <= n ; i ++ )
19      c[i] = c[n-i] = ( n-i+1 ) * c[i-1] / i;
20
21  // 任意组合数
22  #define int long long
23  const int N = 5e5+5 , mod = 1e9+7;
24  int fact[N] , invFact[N];
25
26  int power(int x,int y){...} // 快速幂
27  int inv( int x ){...} // 求逆元, 一般是费马小定理
28
29  int A( int x , int y ){ // x 中选 y 排序
30      return fact[x] * invFact[x-y] % mod;
31  }
32
33  int C( int x , int y ){ // x 中选 y 个
34      return fact[x] * invFact[x-y] % mod * invFact[y] % mod;
35  }
36
37  void init(){
38      fact[0] = 1 , invFact[0] = inv(1);
39      for( int i = 1 ; i < N ; i ++ )
40          fact[i] = fact[i-1] * i % mod , invFact[i] = inv(fact[i]);
41  }
```

4.3.3 公式杂项

$$1^2+2^2+3^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}$$

4.4 线性代数

4.4.1 矩阵加速递推

```
struct matrix {
    static constexpr int mod = 1e9 + 7;
    int x, y;
    vector<vector<int>> v;

    matrix() {}

    matrix(int x, int y) : x(x), y(y) {
        v = vector<vector<int>>(x + 1, vector<int>(y + 1, 0));
    }

    void I() {// 单位化
        y = x;
        v = vector<vector<int>>(x + 1, vector<int>(x + 1, 0));
        for (int i = 1; i <= x; i++) v[i][i] = 1;
        return;
    }

    void display() { // 打印
        for (int i = 1; i <= x; i++)
            for (int j = 1; j <= y; j++)
                cout << v[i][j] << " \n"[j == y];
        return;
    }

    friend matrix operator*(const matrix &a, const matrix &b) { //乘法
        assert(a.y == b.x);
        matrix ans(a.x, b.y);
        for (int i = 1; i <= a.x; i++)
            for (int j = 1; j <= b.y; j++)
                for (int k = 1; k <= a.y; k++)
                    ans.v[i][j] = (ans.v[i][j] + a.v[i][k] * b.v[k][j
                        ]) % mod;
        return ans;
    }

    friend matrix operator^( matrix x , int y ){ // 快速幂
        assert( x.x == x.y );
```

```
38          matrix ans(x.x , x.y);
39          ans.I();//注意一定要先单位化
40          while( y ){
41              if( y&1 ) ans = ans*x;
42              x = x * x , y >>= 1;
43          }
44          return ans;
45      }
46  };
```

例题 Luogo P1939

已知数列 a，满足

$$a_i = \begin{cases} 1 & i \in \{1,2,3\} \\ a_{i-1} + a_{x-3} & x \geq 4 \end{cases}$$

求数列第 n 项对 10^9+7 取模

设计状态阵 $mat_i = \begin{bmatrix} a_i \\ a_{i-1} \\ a_{i-2} \end{bmatrix}$，则 $mat_{i+1} = \begin{bmatrix} a_{i+1} \\ a_i \\ a_{i-1} \end{bmatrix} = \begin{bmatrix} a_i + a_{i-2} \\ a_i \\ a_{i-1} \end{bmatrix}$

可以用待定系数法，加对应项相等解出转移矩阵 $\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$

则 $mat_{3+n} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^n \times mat_3$

4.5 离散数学

第五章　字符串

5.1　字符串哈希

5.1.1　单哈希

这里用unsigned long long的自然溢出做模数

```
1  typedef unsigned long long ull;
2  const int N = 1e6+5 , K = 1e9+7;// 长度 K进制
3  vector<ull> hashP(N);
4
5  void init(){// 初始化
6      hashP[0] = 1;
7      for( int i = 1 ; i < N ; i ++ ) hashP[i] = hashP[i-1] * K;
8  }
9
10 void hashStr( const string & s , vector<ull> & a){ // 计算Hash数组
11     a.resize(s.size()+1);
12     for( int i = 1 ; i <= s.size() ; i ++ )
13         a[i] = ull(a[i-1] * K + s[i-1]);
14 }
15
16 ull hashStr( const string & s ){
17     ull ans = 0;
18     for( auto i : s )
19         ans = ull( ans * K + i );
20     return ans;
21 }
22
23 ull getHash( int l , int r , const vector<ull> & a ){ //计算Hash值
24     return a[r] - a[l-1] *  hashP[r-l+1];
25 }
```

5.1.2 双哈希

这里用unsigned long long的自然溢出和998244353做模数

```
#define int long long
#define mp make_pair
typedef pair<int, int> hashv;

const hashv mod = mp( 1e9+7 , 998244353 );
const hashv base = mp(13331,23333);

hashv operator + ( hashv a , hashv b ) {
    int c1 = a.first + b.first , c2 = a.second + b.second;
    if( c1 >= mod.first ) c1 -= mod.first;
    if( c2 >= mod.second ) c2 -= mod.second;
    return mp( c1 , c2 );
}

hashv operator - ( hashv a , hashv b ) {
    int c1 = a.first-b.first , c2 =a.second-b.second;
    if( c1 < 0 ) c1 += mod.first;
    if( c2 < 0 ) c2 += mod.second;
    return mp( c1 , c2 );
}

hashv operator * ( hashv a , hashv b ) {
    return mp( a.first*b.first%mod.first , a.second*b.second%mod.
       second );
}

const int N = 2e6+5;

vector< hashv > p , hs , ht;

void hashStr( const string &s , vector<hashv> &v ){
    v.resize(s.size()+1);
    for( int i = 1 ; i <= s.size() ; i ++ )
        v[i] = v[i-1] * base + mp( s[i-1] , s[i-1] );
    return;
}
```

```
hashv getHash( int l , int r , const vector<hashv> &v){
    if( l > r ) return mp( 0 , 0 );
    return v[r] - v[l-1] * p[r-l+1];
}
void init( int n ){
    p = vector<hashv>( n+1 ) , p[0] = mp(1,1);
    for( int i = 1 ; i <= n ; i ++ ) p[i] = p[i-1] * base;
}
```

5.2 KMP

首先对字符串首先要求一个前缀函数 $\pi[i]$。$\pi[i]$ 简单来说就是子串 $s[0\dots i]$ 最长的相等的真前缀与真后缀的长度。

```
vector<int> prefix_function(const string &s) {
    int n = s.size();
    vector<int> pi(n);
    for (int i = 1, j; i < n; i++) {
        j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}
```

然后就是 KMP 算法的实现有两种，两种做法效率实际上一样的

```
// pattern 在 text 中出现的位置
vector<int> kmp(const string &text, const string &pattern) {
    string cur = pattern + '#' + text;
    int n = text.size(), m = pattern.size();
    vector<int> v, lps = prefix_function(cur);
    for (int i = m + 1; i <= n + m; i++)
        if (lps[i] == m) v.push_back(i - 2 * m);
    return v;
}
```

除了这样做之外，还有一种做法是求不重复的匹配位置

```
vector<int> kmp(const string &text, const string &pattern) {
```

```
    vector<int> v, lps = prefix_function(pattern);
    for (int i = 0, j = 0; i < text.size(); i++) {
        while (j && text[i] != pattern[j]) j = lps[j - 1];
        if (text[i] == pattern[j]) j++;
        if (j == pattern.size())
            v.push_back(i - j + 1), j = 0;
    }
    return v;
}
```

5.3 Tire

```
struct Trie {
    struct node {
        vector<int> nxt;
        bool exist;
        char val;

        node(char val = '@') : nxt(26, -1), exist(false), val(val) {};
    };

    vector <node> t;

    Trie() : t(1, node()) {};

    void insert(string s) {
        int pos = 0;
        for (auto c: s) {
            int x = c - 'a';
            if (t[pos].nxt[x] == -1)
                t[pos].nxt[x] = t.size(), t.emplace_back(c);
            pos = t[pos].nxt[x];
        }
        return;
    }

    bool find(string s) {
        int pos = 0;
```

```
27          for (auto c: s) {
28              int x = c - 'a';
29              if (t[pos].nxt[x] == -1) return false;
30              pos = t[pos].nxt[x];
31          }
32          return t[pos].exist;
33      }
34  };
```

5.4 最小表示法

5.4.1 循环同构

如果字符串 S 选择一个位置 i 满足

$$S[i...n] + S[1...i-1] = T$$

则称 S 与 T 循环同构

5.4.2 最小表示法

对于一对字符串 A, B，他们在原串中的起始位置分别为 i, j，且前 k 个字符均相同，即

$$S[i...i+k-1] = S[j...j+k-1]$$

若 $S[i+k] > S[j+k]$，则其实下表 $l \in [i, i+k]$ 的字符串均不可能为最优解，因为如果有 $l = i+p$ 则一定有 $j+p$ 字典序更小，所以可以直接把 i 移动到 $i+k+1$ 进行比较。

```
1   int minNotation(const vector<int> &s) {
2       int n = s.size();
3       int i = 0, j = 1;
4       for (int k = 0; k < n && i < n && j < n;) {
5           if (s[(i + k) % n] == s[(j + k) % n]) k++;
6           else {
7               if (s[(i + k) % n] > s[(j + k) % n]) i = i + k + 1;
8               else j = j + k + 1;
9               if (i == j) i++;
10              k = 0;
11          }
12      }
13      return min(i, j);
14  }
```

第六章　动态规划

6.1　线性 DP

6.2　树形 DP

6.2.1　普通树形 DP

> 给一颗 n 个节点有根的树，树上标号 i 的点权值为 h_i。在树上选一些点，要求父节点和子节点不能同时选。问权值和最大是多少?

$f[i][0]$ 表示在 i 子树中选择，不选 i 的最大权值和，$f[i][1]$ 表示选 i 的最大权值和。

对于一对父子 (x, y)，如果选父节点 x，则 y 不能选，`f[x][1] += f[y][0]`。如果不选 x，则 y 随意，`f[x][0] += max(f[y][1] , f[y][0])`。注意选 x 还要加上本身，`f[x][1] += h[x]`。

```
// NC1044A
#include <bits/stdc++.h>
using namespace std;
int read() { ... }

int n, root;
vector<bool> v;
vector<int> h;
vector <vector<int>> e, f;

void dp(int x) {
    f[x][1] = h[x];
    for (auto y: e[x]) {
        dp(y);
        f[x][1] += f[y][0];
        f[x][0] += max(f[y][0], f[y][1]);
    }
}
```

```
int32_t main() {
    n = read();
    v = vector<bool>(n + 1, 0), h = vector<int>(n + 1);
    e = vector < vector < int >> (n + 1);
    f = vector < vector < int >> (n + 1, vector<int>(2));
    for (int i = 1; i <= n; i++) h[i] = read();
    for (int i = 1, x, y; i < n; i++)
        x = read(), y = read(), e[y].push_back(x), v[x] = 1;
    for (int i = 1; i <= n; i++) {
        if (v[i]) continue;
        root = i;
        break;
    }
    dp(root);
    cout << max(f[root][0], f[root][1]);
    return 0;
}
```

6.2.2 背包类树形 DP

给一颗大小为 n 树，树上每个点都有一个点权。选择节点的先决条件是是选择其父节点。最多可以选择 m 个节点，问可选的最大权值是多少。

设 $f[i][j]$ 表示 i 节点子树中选择的不超过 j 个节点的最大权值。从儿子转移过来的过程其实就是一个类似**分组背包**的过程。

```
// NC1044B
void dp(int x) {
    f[x][0] = 0;
    for (auto y: e[x]) {
        dp(y);
        for (int i = m; i >= 0; i--)// 枚举当前节点最大可用背包容积
            for (int j = i; j >= 0; j--)//枚举当子节点最大可用背包容积
                f[x][i] = max(f[x][i], f[x][i - j] + f[y][j]);
    }
    // 这里要给每种容积都加上他本身的权值。
    for (int i = m; x != 0 && i > 0; i--) f[x][i] = f[x][i-1]+val[x];
    return;
}
```

6.2.3 换根 DP

给定一个 n 个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

```
// luogu P3478
#include<bits/stdc++.h>
using namespace std;
#define int long long

int read() {...}

int n;
vector<int> d, f, son;
vector<bool> vis;
vector<vector<int>> e;

void dfs(int x) {
    vis[x] = 1;
    for (auto y: e[x]) {
        if (vis[y]) continue;
        d[y] = d[x] + 1;
        dfs(y);
        son[x] += son[y];
    }
    return;
}

void dp(int x){
    vis[x] = 1;
    for( auto y : e[x] ){
        if( vis[y] ) continue;
        f[y] = f[x] + n - 2*son[y];
        dp(y);
    }
}


```

```
int32_t main() {
    n = read();
    d = vector<int>(n + 1), f = vector<int>(n + 1);
    son = vector<int>(n + 1, 1), son[0] = 0;
    vis = vector<bool>(n + 1);
    e = vector<vector<int>>(n+1);
    for (int i = 1, u, v; i < n; i++)
        u = read(), v = read(), e[u].push_back(v), e[v].push_back(u);
    dfs(1);
    vis = vector<bool>(n + 1);
    for( int i = 1 ; i <= n ; i ++ )
        f[1] += d[i];
    dp(1);
    int val = 0 , res = 0;
    for( int i = 1 ; i <= n ; i ++ )
        if( f[i] > val ) val = f[i] , res = i;
    cout << res;
    return 0;
}
```